﻿4.2 Estructura básica de un programa en Java

﻿public class Holamundo {

/\* Programa holamundo \*/

public static void main(String[] args) {

// Este programa solo muestra un saludo por la pantalla

System.out.println(“Hola, mundo!”); }

﻿En este sencillo programa podemos apreciar varias características importantes de Java: Todo el código Java se debe incrustar dentro de unas estructuras llamadas clases.

Todas las clases deben tener un nombre (“holamundo”) Dentro del código se pueden intercalar comentarios escritos en lenguaje natural, rodeados con los caracteres /\* y \*/, o precedidos de una doble barra: //

Dentro de las clases puede haber muchas cosas. En concreto, al menos una clase debe disponer de un bloque de código (técnicamente hablando, un método) con el nombre main(): por ahí comenzará la ejecución del programa.

Este método es público (se puede ejecutar desde cualquier lugar), es estático (se puede invocar sin instanciar la clase) y no devuelve ningún valor (void) a quien lo ejecutó.

﻿Tipos de datos simples

﻿Los datos, como definimos al principio del capítulo, son representaciones de los objetos del mundo real. Esos objetos pueden ser simples (por ejemplo, la edad de una persona, o el número de trabajadores de una empresa) o complejos (por ejemplo, la flota de camiones de una empresa de transportes).

﻿Nosotros nos referiremos ahora a los tipos de datos simples. Son importantes porque los datos más

﻿complejos se fundamentan en ellos, y es necesario informar a Java de cuáles son los tipos que vamos a usar porque necesita saberlo para reservar los recursos necesarios para almacenarlos (principalmente, memoria RAM)

﻿Pues bien, atendiendo a cómo el lenguaje maneja los tipos de datos, podemos distinguir:

﻿Lenguajes con tipado dinámico: una misma variable puede cambiar de tipo a lo largo de la ejecución del programa. Ejemplos: Perl, Python, Lisp, JavaScript, PHP.

Lenguajes con tipado estático: una variable, una vez que es asignada a un tipo de datos, no puede cambiar de tipo. Es menos flexible que el tipado dinámico, pero también más eficiente. Ejemplos: C, C++, Java, Basic, Pascal.

Lenguajes débilmente tipados: no hacen comprobaciones exhaustivas de tipos de datos. Así, permiten manipular los datos de determinado tipo como si fueran de otro tipo. Por ejemplo, ﻿un dato de tipo carácter puede manipularse, si al programador le conviene, como un dato numérico (ya que en el fondo los caracteres son números). Esto puede provocar resultados extraños si el programador comete un error, pero proporciona mucha flexibilidad si se usa correctamente.

Lenguajes fuertemente tipados: comprueban exhaustivamente que las variables de cada tipo sólo se usan conforme a lo que ese tipo permite. Por ejemplo, no se permitirá realizar una operación de suma con caracteres. Son menos flexibles y, además, más ineficientes, puesto que deben realizar comprobaciones de tipo en tiempo de ejecución, es decir, deben introducir código máquina adicional para hacer las comprobaciones de tipo.

A cambio, suelen generar ﻿programas mucho más robustos y tolerantes a fallos. Es habitual confundir el tipado estático con el tipado fuerte, y el dinámico con el débil. En realidad, son categorías complementarias que se pueden mezclar: existen lenguajes con tipado estático y débil (como C) y otros con tipado dinámico y fuerte (como Visual Basic).

**Por su parte, Java es un lenguaje con tipado estático y fuerte. Esto significa que las variables no pueden cambiar de tipo una vez asignado, y que se hacen comprobaciones estrictas sobre los tipos en tiempo de ejecución.**

﻿5.2 Tipos de datos primitivos en Java

﻿Cada tipo de datos, además, tiene asociado un conjunto de operaciones para manipularlos. Cada tipo de datos, insistimos, dispone de una representación interna diferente en el ordenador; por eso es importante distinguir entre tipos de datos a la hora de programar.

**Los tipos primitivos de Java son: Números enteros, Números reales, Caracteres, Lógicos**

﻿Los tipos enteros primitivos en Java son:

byte: entero de 8 bits con signo.

short: entero de 16 bits con signo.

int: entero de 32 bits con signo.

long: entero de 64 bits con signo.

﻿Java dispone de dos tipos primitivos para manejar números reales:

float: coma flotante de 32 bits (1 bit de signo, 8 de exponente y 24 de mantisa)

double: coma flotante de 64 bits (1 bit de signo, 11 de exponente y 52 de mantisa)

﻿Overflow

﻿Cuando se realizan operaciones con números (tanto enteros como reales), es posible que el resultado de una de ellas dé lugar a un número fuera del rango máximo permitido.

﻿En estos casos, estamos ante un caso extremo denominado overflow o desbordamiento. Los ordenadores pueden reaccionar de forma diferente ante este problema, dependiendo del sistema operativo y del lenguaje utilizado. Algunos lo detectan como un error de ejecución del programa, mientras que otros lo ignoran, convirtiendo el número desbordado a un número dentro del rango permitido pero que, obviamente, no será el resultado correcto de la operación, por lo que el programa probablemente fallará. **En el caso de Java, la JVM proporcionará un error en tiempo de ejecución si se produce un desbordamiento. Ese error puede capturarse mediante una excepción para tratarlo adecuadamente. Veremos más adelante como hacer todo esto.**

﻿﻿Caracteres y cadenas

﻿El tipo de dato carácter sirve para representar datos alfanuméricos. El conjunto de elementos que puede representar está estandarizado según diferentes tablas de código (ASCII o Unicode). El estándar más antiguo es ASCII, que consiste en una combinación de 7 u 8 bits asociada a un carácter alfanumérico concreto, pero que está siendo sustituido por sus limitaciones en la representación de caracteres no occidentales.

﻿**Java proporciona el tipo char, de 16 bits, para manejar caracteres**.

﻿Los caracteres válidos que pueden almacenarse en una variable tipo char son: Las letras minúsculas: 'a', 'b', 'c' ... 'z' Las letras mayúsculas: 'A', 'B', 'C' ... 'Z' Los dígitos: '1', '2', '3' ... Caracteres especiales o de otros idiomas: '$', '%', '¿', '!' , 'ç'... Nótese que no es lo mismo el valor entero 3 que el carácter '3'. Para distinguirlos, usaremos siempre comillas para escribir los caracteres.

﻿Datos lógicos

﻿El tipo dato lógico, también llamado booleano en honor a George Boole (un matemático británico que desarrolló en el siglo XIX una rama del álgebra llamada lógica o de Boole) es un dato que sólo puede tomar un valor entre dos posibles. Esos dos valores son: Verdadero (en inglés, true) Falso (en inglés, false) ﻿con el número 1 (o con cualquier número distinto de 0).

﻿En Java, los datos booleanos se manejan mediante el tipo boolean.

﻿Conversiones de tipo (casting)

﻿Java es un lenguaje fuertemente tipado, por lo que suele reaccionar mal ante el intento de mezclar tipos de datos distintos en la misma expresión. En general, en estos casos se puede hablar de dos tipos de conversión de datos:

﻿A) Conversiones implícitas: se realizan de forma automática al mezclar tipos de datos. En Java solo puede hacerse si la variable receptora del resultado tiene más precisión que las variables situadas en la expresión. Por ejemplo, puede asignarse un int a un long, pero no al revés:

﻿long a = 1, b = 7;

int x = 3, y = 6;

a = x; // Esta asignación funcionará

y = b; // Esta asignación fallará

﻿Conversiones explícitas: el programador especifica mediante el código la conversión de un tipo en otro, indicando el nuevo tipo entre paréntesis durante a asignación. Este proceso se denomina casting y se muestra en el siguiente ejemplo:

﻿int x = 5;

byte y;

﻿y = (byte)x; // La variable entera x se convertirá a byte

﻿Como un int puede contener números más grandes que un byte, esta última conversión puede suponer la pérdida de parte de la información y debe ser, en general, evitada.

﻿5.3 Operaciones con datos

﻿5.3.1 Operaciones aritméticas

﻿Operación Operador

suma +

resta –

multiplicación \*

división /

módulo (resto) % (﻿Sirve para calcular el resto de la división entera).

﻿Hay, además, otros dos operadores aritméticos muy utilizados: Operador incremento (++): Se utiliza para aumentar en una unidad el valor de una variable numérica entera.

**Por ejemplo, x++ es equivamente a x = x + 1.**

**﻿Operador decremento: (–). Se utiliza para disminuir en una unidad el valor de una variable numérica entera. La expresión x-- es equivamente a x = x – 1.**

﻿5.3.2 Operaciones lógicas (o booleanas)

﻿Estas operaciones sólo pueden dar como resultado verdadero o falso, es decir, su resultado debe ser un valor lógico. Hay dos tipos de operadores que se utilizan en estas operaciones: los operadores de relación y los operadores lógicos.

﻿1) Operadores de relación. Son los siguientes:

﻿Operación Operador

menor que <

mayor que >

igual que ==

menor o igual que <=

mayor o igual que >=

﻿distinto de !=

﻿Los operadores de relación se pueden usar con todos los tipos de datos simples: entero, real, carácter o lógico. El resultado será verdadero si la relación es cierta, o falso en caso contrario.

﻿En cuanto a los datos lógicos, se considera, por convenio, que "falso" es menor que "verdadero". Por lo tanto:

Tabla

Descripción generada automáticamente

﻿2) Operadores lógicos. Los operadores lógicos son and (y), or (o) y not (no). Sólo se pueden emplear con tipos de datos lógicos.

﻿El operador and, que también podemos llamar y, se escribe en Java como &&, y da como resultado verdadero sólo si los dos operandos son verdaderos:

﻿El operador or (también nos vale o) da como resultado verdadero cuando al menos uno de los dos operandos es verdadero. En Java se escribe | |

﻿El operador not (o no), que se escribe !, es uno de los escasos operadores que sólo afectan a un operando (operador monario), no a dos (operador binario). El resultado es la negación del valor del operando, es decir, que le cambia el valor de verdadero a falso y viceversa:

﻿5.3.4 Prioridad de los operadores

﻿La prioridad de cálculo respeta las reglas generales del álgebra. Así, por ejemplo, la división y la multiplicación tienen más prioridad que la suma o la resta. Pero el resto de prioridades pueden diferir de manera imporante de un lenguaje de programación a otro.

﻿La prioridad del cálculo se puede alterar usando paréntesis, como en el álgebra convencional. Los paréntesis se pueden anidar tantos niveles como sean necesarios. Por supuesto, a igualdad de prioridad entre dos operadores, la operación se calcula de izquierda a derecha, en el sentido de la lectura de los operandos.

﻿5.3.5 ¿Y las operaciones más complejas?

﻿Además de todas estas operaciones artiméticas, lógicas y relacionales, los lenguajes de programación disponen de mecanismos para realizar operaciones más complejas con los datos, como, por ejemplo, calcular raices cuadradas, logaritmos, senos, cosenos, redondeo de números reales, etc.

﻿La ejecución de estos métodos predefinidos requiere el uso de una clase de la biblioteca estándar llamada Math, porque en ella se agrupan todos los métodos de índole matemática. Al método se le pasarán los parámetros necesarios para que haga sus cálculos, y nos devolverá el resultado.

﻿Estas son algunos de los métodos que nos pueden resultar muy útiles hasta que veamos en profundidad la biblioteca de clases:

Tabla

Descripción generada automáticamenteTabla

Descripción generada automáticamente

﻿5.4 Constantes y variables

﻿Un dato constante (o, simplemente, "una constante") es un dato de un programa cuyo valor no cambia durante la ejecución. Por el contrario, un dato variable (o, simplemente, "una variable") es un dato cuyo valor sí cambia en el transcurso del programa.

﻿5.4.1 Identificadores

﻿A las constantes y variables se les asigna un identificador alfanumérico, es decir, un nombre. Por lo tanto, es necesario disinguir entre el identificador de una variable y su valor. ﻿Por ejemplo, una variable ﻿llamada x puede contener el valor 5. En este caso, x es el identificador y 5 el valor de la variable.

﻿Los identificadores o nombres de variable deben cumplir ciertas reglas que, aunque varían de un lenguaje a otro, podemos resumir en que:

Deben empezar por una letra y, en general, no contener símbolos especiales excepto el subrayado ("\_")

No deben coincidir con alguna palabra reservada del lenguaje, tales como class o function.

﻿Las constantes también deben tener un identificador. Por cuestión de estilo, en Java los identificadores de las constanes se escriben en MAYÚSCULA, y, los de las variables, en minúscula. Pero solo es una regla de estilo: conveniente, pero no imprescindible. En cada organización pueden tener sus propias reglas.

﻿5.4.2 Declaración y asignación

﻿Las variables, si son simples, tienen que ser de un tipo de datos determinado, es decir, debemos indicar explícitamente qué tipo de datos va a almacenar a lo largo del programa. Y, si son complejas, deben ser instancias de una clase concreta, aunque esto ya lo explicaremos más adelante.

Por ahora, recuerda esto: es imprescindible indicar cuál va a ser el identificador de la variable, y qué tipo de datos va a almacenar.

A esto se le llama declarar la variable.

Una declaración de variables en Java será algo así:

int x;

double y;

char z;

﻿x, y, z son los identificadores de variable.

﻿El compilador debe saber de antemano qué variables va a usar el programa y de qué tipo son para poder asignarles la memoria necesaria cuando comience la ejecución y para saber qué operaciones nos permitirá hacer con esa variable.

﻿Para adjudicar un valor a una variable, se emplean las sentencias de asignación, que tienen esta forma:

x = 5;

y = 7.445;

z = 'J';

﻿A partir de la asignación, pueden hacerse operaciones con las variables exactamente igual que se harían con datos. Por ejemplo, la operación x + x daría como resultado 10. A lo largo del programa, la misma variable x puede contener otros valores ﻿(siempre de tipo entero) y utilizarse para otras operaciones.

﻿En cambio, las constantes son valores que nunca cambian. Sólo se les puede asignar valor una vez, ya que, por su propia naturaleza, son invariables a lo largo del programa.

﻿En Java, **una constante se distingue con el modificador *final*** colocado antes de la declaración, y la asignación debe hacerse en ese mismo momento.

Por ejemplo: final PI = 3.141592;

﻿5.4.3 El modificador static

﻿El modificador static significa que esa variable o constante solo se crearán una vez en toda la ejecución del programa, aunque aparezcan declaradas varias veces.

﻿static int x;

final static PI = 3.141592;

﻿5.4.4 Vida y ámbito de las variables

﻿Las variables en Java son, por definición, todas locales al bloque en el que se han declarado, entendiendo por bloque a cualquier conjunto de instrucciones enmarcado entre una llave { y otra llave }

﻿Expresiones

﻿Una expresión es una combinación de constantes, variables, operadores, métodos y expresiones literales, tales como 5 o 28.33. Es decir, se trata de operaciones aritméticas o lógicas en las que se pueden combinar todos los elementos vistos hasta ahora, y que resultan evaluables por la máquina para proporcionar un resultado único.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-﻿Ejercicio 1.1: Instalación del JDK

Vozmediano, A. M.. Java: (p. 65). Edición de Kindle. -----------------

Consola:

$ /usr/libexec/java\_home -v17 (para saber la ruta de inicio de java)

**/Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home**

anagarcia@MacBook-Air-de-Ana ~ % cd /Library/Java/JavaVirtualMachines

anagarcia@MacBook-Air-de-Ana JavaVirtualMachines % vim ~/.bash\_profile

HOLA MUNDO

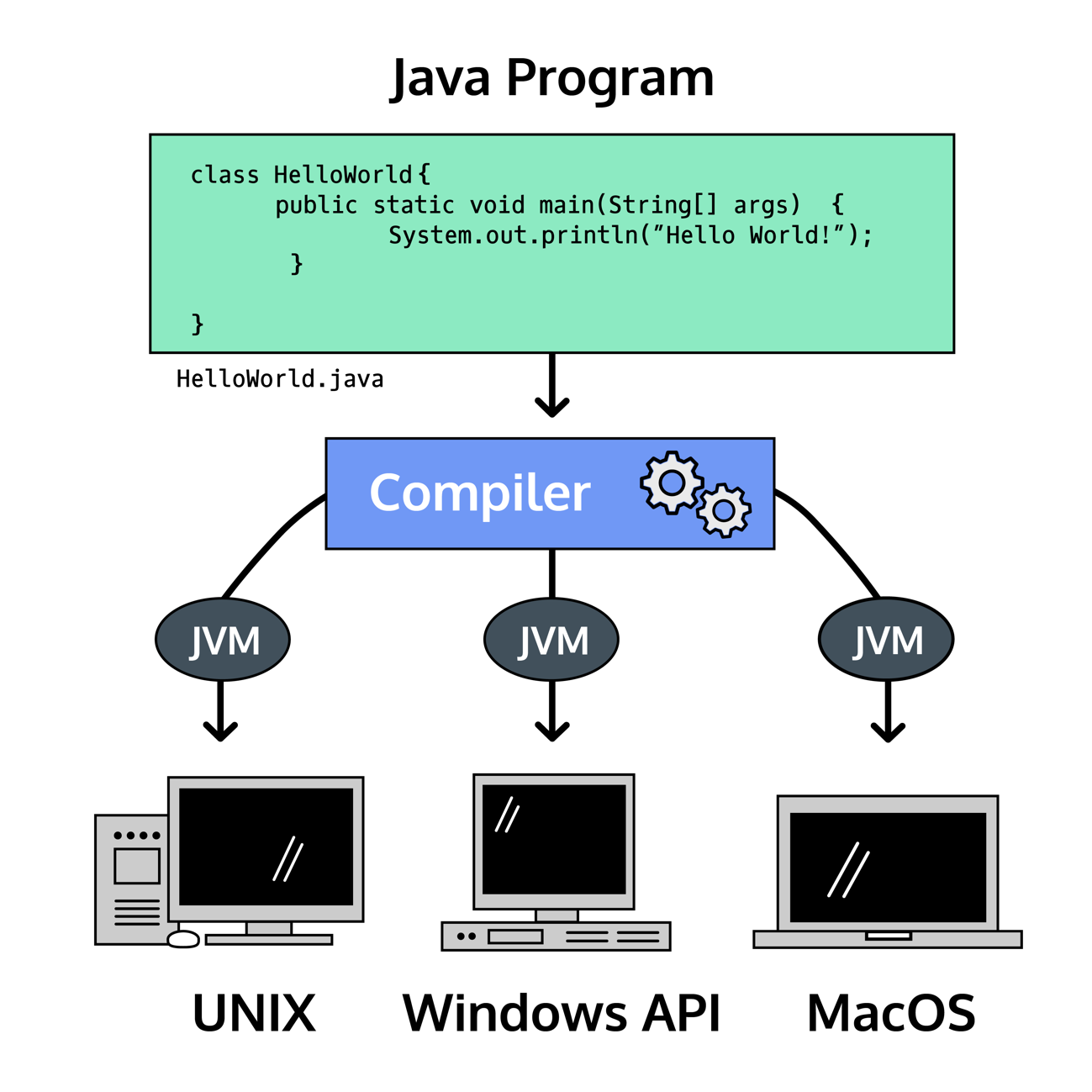
Introducción a Java

¡Bienvenido al mundo de la programación Java!

Los lenguajes de programación permiten a los humanos escribir instrucciones que una computadora puede ejecutar. Con instrucciones precisas, las computadoras coordinan aplicaciones y sistemas que ejecutan el mundo moderno.

Sun Microsystems lanzó el lenguaje de programación Java en 1995. Java es conocido por ser simple, portátil, seguro y robusto. Aunque fue lanzado hace más de veinte años, Java sigue siendo uno de los lenguajes de programación más populares en la actualidad.

Una de las razones por las que a la gente le encanta Java es la máquina virtual Java, que garantiza que el mismo código Java se pueda ejecutar en diferentes sistemas operativos y plataformas. El lema de Sun Microsystems para Java era "escribir una vez, ejecutar en todas partes".



Los lenguajes de programación se componen de sintaxis, las instrucciones específicas que comprende Java. Escribimos sintaxis en archivos para crear programas, que son ejecutados por la computadora para realizar la tarea deseada.

Comencemos con el saludo universal para un lenguaje de programación. Exploraremos la sintaxis en el próximo ejercicio.

**Classes**

Una class representa un solo concepto.

Un programa Java debe tener una clase cuyo nombre sea el mismo que el nombre del archivo del programa.

En el ejemplo, la clase Person debe declararse en un archivo de programa llamado Person.java.

### main() Method

En Java, cada aplicación debe contener un método main (), que es el punto de entrada para la aplicación. Todos los demás métodos se invocan desde el método main ().

La firma del método es public static void main (String [] args) {}. Acepta un solo argumento: una matriz de elementos de tipo String.

public class Person {  
    
  public static void main(String[] args) {  
      
    System.out.println("Hello, world!");  
  }  
    
}

### Print Line

System.out.println () puede imprimir en la consola:

El sistema es una clase de la biblioteca principal proporcionada por Java out es un objeto que controla la salida println () es un método asociado con ese objeto que recibe un solo argumento

System.out.println("Hello, world!");  
// Output: Hello, world!

Comentarios

Los comentarios son fragmentos de texto que el compilador ignora. Se utilizan para aumentar la legibilidad de un programa.

Los comentarios de una sola línea se crean mediante //. Los comentarios de varias líneas se crean comenzando con / \* y terminando con \* /.

// I am a single line comment!  
/\*  
And I am a   
multi-line comment!  
\*/

Espacio en blanco

Se ignoran los espacios en blanco, incluidos los espacios y las nuevas líneas, entre las declaraciones.

System.out.println("Example of a statement");  
System.out.println("Another statement");  
// Output:  
// Example of a statement  
// Another statement

Compilando Java

En Java, cuando compilamos un programa, cada clase individual se convierte en un archivo .class, que se conoce como código de bytes.

La JVM (máquina virtual Java) se utiliza para ejecutar el código de bytes.

# Compile the class file:  
javac hello.java  
# Execute the compiled file:  
java hello

Declaraciones

En Java, una declaración es una línea de código que ejecuta una tarea y termina con un ;.

System.out.println("Java Programming ☕️");

**Variables**

**tipo de datos booleano**

En Java, el tipo de datos primitivo booleano se utiliza para almacenar un valor, que puede ser verdadero o falso.

boolean result = true;  
boolean isMarried = false;

**Strings**

Una cadena en Java es un objeto que contiene varios caracteres. No es un tipo de datos primitivo.

Se puede crear un String colocando caracteres entre un par de comillas dobles (").

Para comparar String, se debe utilizar el método equals () en lugar del comparador de igualdad primitivo ==.

// Creating a String variable  
String name = "Bob";  
  
// The following will print "false" because strings are case-sensitive  
System.out.println(name.equals("bob"));

### int Data Type

En Java, el tipo de datos int se usa para almacenar valores enteros. Esto significa que puede almacenar todos los números enteros positivos y negativos y el cero.

int num1 = 10;   // positive value  
int num2 = -5;   // negative value  
int num3 = 0;    // zero value  
int num4 = 12.5; // not allowed

### char Data Type

En Java, char se usa para almacenar un solo carácter. El carácter debe ir entre comillas simples.

char answer = 'y';

### Primitive Data Types

Los tipos de datos más básicos de Java se conocen como tipos de datos primitivos y se encuentran en el sistema de forma predeterminada.

**Los tipos disponibles son los siguientes:**

* int
* char
* boolean
* byte
* long
* short
* double
* float
* null es otro, pero solo puede almacenar el valor null.

int age = 28;   
char grade = 'A';  
boolean late = true;  
byte b = 20;  
long num1 = 1234567;  
short no = 10;  
float k = (float)12.5;  
double pi = 3.14;

### Static Typing

En Java, el tipo de variable se comprueba en tiempo de compilación. Esto se conoce como escritura estática. Tiene la ventaja de detectar los errores en tiempo de compilación en lugar de en tiempo de ejecución.

Las variables deben declararse con el tipo de datos apropiado o el programa no se compilará.

int i = 10;         // type is int  
char ch = 'a';      // type is char  
  
j = 20;             // won't compile, no type is given  
char name = "Lil";  // won't compile, wrong data type

### final Keyword

El valor de una variable no se puede cambiar si la variable se declaró utilizando la palabra clave final.

Tenga en cuenta que la variable debe recibir un valor cuando se declara como final. las variables finales no se pueden cambiar; cualquier intento de hacerlo resultará en un mensaje de error.

// Value cannot be changed:  
final double PI = 3.14;

### double Data Type

El tipo primitivo double se utiliza para contener valores decimales.

double PI = 3.14;  
double price = 5.75;

**Operaciones matemáticas**

Las operaciones matemáticas básicas se pueden aplicar a tipos de datos int, double y float:

+ adición

- resta

\* multiplicación

/ división

% modulo (produce el resto)

Estas operaciones no son compatibles con otros tipos de datos.

int a = 20;  
int b = 10;  
int result;  
result = a + b;  // 30  
result = a - b;  // 10  
result = a \* b;  // 200  
result = a / b;  // 2  
result = a % b;  // 0

### Operadores de comparación

### Los operadores de comparación se pueden utilizar para comparar dos valores:

### > mayor que

### <menor que

### > = mayor o igual que

### <= menor o igual que

### == igual a

### ! = no es igual a

### Son compatibles con tipos de datos primitivos y el resultado de una comparación es un valor booleano verdadero o falso.

int a = 5;  
int b = 3;  
  
boolean result = a > b;  
// result now holds the boolean value true

**Operadores de asignación compuesta**

Los operadores de asignación compuesta se pueden usar para cambiar y reasignar el valor de una variable usando una línea de código. Los operadores de asignación compuesta incluyen + =, - =, \* =, / = y% =.

int number = 5;  
number += 3; // Value is now 8  
number -= 4; // Value is now 4  
number \*= 6; // Value is now 24  
number /= 2; // Value is now 12  
number %= 7; // Value is now 5

**Operadores de incremento y decremento**

El operador de incremento, (++), puede aumentar el valor de una variable basada en números en 1, mientras que el operador de disminución, (-), puede disminuir el valor de una variable en 1.

int numApples = 5;  
numApples++; // Value is now 6  
  
int numOranges = 5;  
numOranges--; // Value is now 4

**Orden de operaciones**

El orden en el que se evalúa una expresión con múltiples operadores está determinado por el orden de las operaciones: paréntesis -> multiplicación -> división -> módulo -> suma -> resta.

**Object-Oriented Java**

Estado y comportamiento de los objetos Java

En Java, las instancias de una clase se conocen como objetos. Cada objeto tiene estado y comportamiento en forma de campos de instancia y métodos, respectivamente.

public class Person {  
  // state of an object  
  int age;  
  String name;  
    
  // behavior of an object  
  public void set\_value() {  
    age = 20;  
    name = "Robin";  
  }  
  public void get\_value() {  
    System.out.println("Age is " + age);  
    System.out.println("Name is " + name);  
  }  
    
  // main method  
  public static void main(String [] args) {  
    // creates a new Person object  
    Person p = new Person();   
      
    // changes state through behavior  
    p.set\_value();  
  }  
}

**Método de constructor en Java**

Las clases de Java contienen un método constructor que se utiliza para crear instancias de la clase.

El constructor lleva el nombre de la clase. Si no se define ningún constructor, se utiliza un constructor vacío predeterminado.

public class Maths {  
  public Maths() {  
    System.out.println("I am constructor");  
  }  
  public static void main(String [] args) {  
    System.out.println("I am main");  
    Maths obj1 = new Maths();  
  }  
}

**Instancia de Java**

Las instancias de Java son objetos que se basan en clases. Por ejemplo, Bob puede ser una instancia de la clase Person.

Cada instancia tiene acceso a su propio conjunto de variables que se conocen como campos de instancia, que son variables declaradas dentro del alcance de la instancia. Los valores, por ejemplo, los campos se asignan dentro del método del constructor.

public class Person {  
  int age;  
  String name;  
    
  // Constructor method  
  public Person(int age, String name) {  
    this.age = age;  
    this.name = name;  
  }  
    
  public static void main(String[] args) {  
    Person Bob = new Person(31, "Bob");  
    Person Alice = new Person(27, "Alice");  
  }  
}

**Creando una nueva instancia de clase en Java**

En Java, usamos la palabra clave new seguida de una llamada al constructor de la clase para crear una nueva instancia de una clase.

El constructor se puede utilizar para proporcionar valores iniciales a los campos de instancia.

public class Person {  
  int age;  
  // Constructor:  
  public Person(int a) {  
    age = a;  
  }  
    
  public static void main(String [] args) {  
    // Here, we create a new instance of the Person class:  
    Person p = new Person(20);  
    System.out.println("Age is " + p.age); // Prints: Age is 20  
  }  
}

Notación de puntos de Java

En el lenguaje de programación Java, usamos . para acceder a las variables y métodos de un objeto o una Clase.

Esto se conoce como notación de puntos y la estructura se ve así: instanceOrClassName.fieldOrMethodName

public class Person {  
  int age;  
    
  public static void main(String [] args) {  
    Person p = new Person();  
      
    // here we use dot notation to set age  
    p.age = 20;   
      
    // here we use dot notation to access age and print  
    System.out.println("Age is " + p.age);  
    // Output: Age is 20  
  }  
}

Tipos de datos de referencia

Una variable con un tipo de datos de referencia tiene un valor que hace referencia a la dirección de memoria de una instancia. Durante la declaración de variable, el nombre de la clase se utiliza como tipo de variable.

public class Cat {  
  public Cat() {  
    // instructions for creating a Cat instance  
  }    
   
  public static void main(String[] args) {  
    // garfield is declared with reference data type `Cat`   
    Cat garfield = new Cat();  
    System.out.println(garfield); // Prints: Cat@76ed5528  
  }  
}

**Firmas de constructores**

Una clase puede contener varios constructores siempre que tengan diferentes valores de parámetro. Una firma ayuda al compilador a diferenciar entre los diferentes constructores.

Una firma se compone del nombre del constructor y una lista de sus parámetros.

// The signature is `Cat(String furLength, boolean hasClaws)`.  
public class Cat {  
  String furType;  
  boolean containsClaws;  
   
  public Cat(String furLength, boolean hasClaws) {  
    furType = furLength;  
    containsClaws = hasClaws;  
  }  
  public static void main(String[] args) {  
    Cat garfield = new Cat("Long-hair", true);  
  }  
}

**Valores nulos**

null es un valor especial que denota que un objeto tiene una referencia vacía.

public class Bear {  
  String species;  
  public Bear(String speciesOfBear;) {  
    species = speciesOfBear;  
  }  
   
  public static void main(String[] args) {  
    Bear baloo = new Bear("Sloth bear");   
    System.out.println(baloo); // Prints: Bear@4517d9a3  
    // set object to null  
    baloo = null;  
    System.out.println(baloo); // Prints: null  
  }  
}

**El cuerpo de un método Java**

En Java, usamos llaves {} para encerrar el cuerpo de un método.

Las declaraciones escritas dentro de {} se ejecutan cuando se llama a un método.

public class Maths {  
  public static void sum(int a, int b) { // Start of sum  
    int result = a + b;  
    System.out.println("Sum is " + result);  
  } // End of sum  
    
    
  public static void main(String [] args) {  
    // Here, we call the sum method  
    sum(10, 20);  
    // Output: Sum is 30  
  }  
}

**Parámetros de método en Java**

En Java, los parámetros se declaran en una definición de método. Los parámetros actúan como variables dentro del método y mantienen el valor que se pasó. Se pueden utilizar dentro de un método con fines de impresión o cálculo.

En el ejemplo, ayb son dos parámetros que, cuando se llama al método, tienen el valor 10 y 20 respectivamente.

public class Maths {  
  public int sum(int a, int b) {  
    int k = a + b;  
    return k;  
  }  
    
  public static void main(String [] args) {  
    Maths m = new Maths();  
    int result = m.sum(10, 20);  
    System.out.println("sum is " + result);  
    // prints - sum is 30  
  }  
}

**Variables de Java dentro de un método**

Las variables de Java definidas dentro de un método no se pueden usar fuera del alcance de ese método.

//For example, `i` and `j` variables are available in the `main` method only:  
   
public class Maths {  
  public static void main(String [] args) {  
    int i, j;  
    System.out.println("These two variables are available in main method only");  
  }  
}

**Devolver información de un método Java**

Un método Java puede devolver cualquier valor que se pueda guardar en una variable. El valor devuelto debe coincidir con el tipo de retorno especificado en la firma del método.

El valor se devuelve utilizando la palabra clave return.

public class Maths {   
    
  // return type is int  
  public int sum(int a, int b) {  
    int k;  
    k = a + b;  
      
    // sum is returned using the return keyword  
    return k;  
  }  
    
  public static void main(String [] args) {  
    Maths m = new Maths();  
    int result;  
    result = m.sum(10, 20);  
    System.out.println("Sum is " + result);  
    // Output: Sum is 30  
  }  
}

**Declarar un método**

Las declaraciones de métodos deben definir la siguiente información del método: alcance (privado o público), tipo de retorno, nombre del método y cualquier parámetro que reciba.

// Here is a public method named sum whose return type is int and has two int parameters a and b  
public int sum(int a, int b) {  
  return(a + b);  
}

Definición de métodos

Si tuviéramos que definir un método checkBalance () para el ejemplo de cuenta de ahorros del que hablamos anteriormente, se vería así:

public void checkBalance(){  
  System.out.println("Hello!");  
  System.out.println("Your balance is " + balance);  
}

La primera línea, public void checkBalance (), es la declaración del método. Le da al programa alguna información sobre el método:

* **public** significa que otras clases pueden acceder a este método.
* La palabra clave **void** significa que no hay una salida específica del método. Veremos métodos que no son nulos más adelante en esta lección, pero por ahora, todos nuestros métodos serán nulos.
* checkBalance () es el nombre del método.

Cada método tiene su propia firma de método única que se compone del nombre del método y su tipo de parámetro. En este ejemplo, la firma del método es checkBalance ().

Las dos declaraciones de impresión están dentro del cuerpo del método, que está definido por las llaves: {y}.

¡Todo lo que podamos hacer en nuestro método main (), lo podemos hacer en otros métodos! Todas las herramientas de Java que conoce, como los operadores matemáticos y de comparación, se pueden utilizar para crear métodos interesantes y útiles.

checkBalance () se considera un método no estático porque su firma no incluye la palabra clave static como lo hace el método main (). Aprenderemos más sobre los métodos no estáticos más adelante en este curso.

public class Dog {

// instance field

String breed;

// constructor method

boolean hasOwner;

int age;

public Dog(String dogBreed, boolean dogOwned, int dogYears) {

   /\*   
    value of parameter dogBreed   
    assigned to instance field breed  
    \*/

System.out.println("Constructor invoked!");

breed = dogBreed;

hasOwner = dogOwned;

age = dogYears;

}

public static void main(String[] args) {

   /\*   
    create instance:   
    use 'new' operator and invoke constructor  
    \*/

System.out.println("Main method started");

Dog fido = new Dog("poodle", false, 4);

    /\*   
    fields are accessed using:   
    the instance name, `.` operator, and the field name.  
    \*/

Dog nunzio = new Dog("shiba inu", true, 12);

boolean isFidoOlder = fido.age > nunzio.age;

int totalDogYears = nunzio.age + fido.age;

System.out.println("Two dogs created: a " + fido.breed + " and a " + nunzio.breed);

System.out.println("The statement that fido is an older dog is: " + isFidoOlder);

System.out.println("The total age of the dogs is: " + totalDogYears);

System.out.println("Main method finished");

  fido.breed;  
    // "poodle"

}

}

Definición de un método (*Defining a method*): los métodos tienen una firma de método que declara su tipo de retorno (return), nombre (name) y parámetros (parameters).

Llamar a un método (*Calling a method*): los métodos se invocan con un . y ()

Parámetros (*Parameters*): las entradas al método y sus tipos se declaran entre paréntesis en la firma del método.

Cambiar campos de instancia (*Changing Instance Fields*): los métodos se pueden utilizar para cambiar el valor de un campo de instancia.

Alcance (Scope): las variables solo existen dentro del dominio en el que se crean.

Devolución (Return): el tipo de las variables que se generan se declaran en la firma del método.

**Condicionales y flujo de control**

if Statement

Una sentencia if ejecuta un bloque de código cuando una expresión booleana especificada se evalúa como verdadera.

if (true) {  
    System.out.println("This code executes");  
}  
// Prints: This code executes  
  
if (false) {  
    System.out.println("This code does not execute");  
}  
// There is no output for the above statement

**else Statement**

La instrucción else ejecuta un bloque de código cuando la condición dentro de la instrucción if es falsa. La instrucción else es siempre la última condición.

boolean condition1 = false;  
  
if (condition1){  
    System.out.println("condition1 is true");  
}  
else{  
    System.out.println("condition1 is not true");  
}  
// Prints: condition1 is not true

### else if Statements

Las declaraciones else-if se pueden encadenar juntas para verificar múltiples condiciones. Una vez que una condición es verdadera, se ejecutará un bloque de código y se saldrá de la declaración condicional.

Puede haber varias declaraciones else-if en una sola declaración condicional.

int testScore = 76;  
char grade;  
  
if (testScore >= 90) {  
  grade = 'A';  
} else if (testScore >= 80) {  
  grade = 'B';  
} else if (testScore >= 70) {  
  grade = 'C';  
} else if (testScore >= 60) {  
  grade = 'D';  
} else {  
  grade = 'F';  
}  
  
System.out.println("Grade: " + grade); // Prints: C

**Declaraciones condicionales anidadas**

Una declaración condicional anidada es una declaración condicional anidada dentro de otra declaración condicional. La declaración condicional externa se evalúa primero; si la condición es verdadera, entonces se evaluará la declaración condicional anidada.

boolean studied = true;  
boolean wellRested = true;  
  
if (wellRested) {  
  System.out.println("Best of luck today!");    
  if (studied) {  
    System.out.println("You are prepared for your exam!");  
  } else {  
    System.out.println("Study before your exam!");  
  }  
}  
  
// Prints: Best of luck today!  
// Prints: You are prepared for your exam!

### AND Operator

### El operador lógico AND está representado por &&. Este operador devuelve verdadero si las expresiones booleanas en ambos lados del operador son verdaderas; de lo contrario, devuelve falso.

System.out.println(true && true); // Prints: true  
System.out.println(true && false); // Prints: false  
System.out.println(false && true); // Prints: false  
System.out.println(false && false); // Prints: false

### The OR Operator

### El operador lógico OR está representado por ||. Este operador devolverá verdadero si al menos una de las expresiones booleanas que se están comparando tiene un valor verdadero; de lo contrario, devolverá falso.

System.out.println(true || true); // Prints: true  
System.out.println(true || false); // Prints: true  
System.out.println(false || true); // Prints: true  
System.out.println(false || false); // Prints: false

### NOT Operator

### El operador lógico NOT está representado por !. Este operador niega el valor de una expresión booleana.

boolean a = true;  
System.out.println(!a); // Prints: false  
  
System.out.println(!true) // Prints: true

**Operadores condicionales: orden de evaluación**

Si una expresión contiene varios operadores condicionales, el orden de evaluación es el siguiente: Expresiones entre paréntesis -> NOT -> AND -> OR.

boolean foo = true && (!false || true); // true  
/\*   
(!false || true) is evaluated first because it is contained within parentheses.   
  
Then !false is evaluated as true because it uses the NOT operator.   
  
Next, (true || true) is evaluation as true.   
  
Finally, true && true is evaluated as true meaning foo is true. \*/

El orden de evaluación cuando se trata de operadores condicionales es el siguiente:

1. Conditions placed in parentheses - ()
2. NOT - !
3. AND - &&
4. OR - ||

**Arrays and ArrayLists**

### Index

### Un índice se refiere a la posición de un elemento dentro de una matriz.

### El índice de una matriz comienza en 0 y sube hasta uno menos que la longitud total de la matriz.

int[] marks = {50, 55, 60, 70, 80};  
  
System.out.println(marks[0]);  
// Output: 50  
  
System.out.println(marks[4]);  
// Output: 80

### Arrays

### En Java, una matriz se usa para almacenar una lista de elementos del mismo tipo de datos.

### Las matrices tienen un tamaño fijo y sus elementos están ordenados.

// Create an array of 5 int elements  
int[] marks = {10, 20, 30, 40, 50};

### Array creation in Java

### En Java, una matriz se puede crear de las siguientes formas:

### Usando la notación {}, agregando cada elemento de una vez.

### Usando la nueva palabra clave y asignando cada posición de la matriz individualmente.

int[] age = {20, 21, 30};  
  
int[] marks = new int[3];  
marks[0] = 50;   
marks[1] = 70;  
marks[2] = 93;

### Cambiar el valor de un elemento

### Para cambiar el valor de un elemento, seleccione el elemento a través de su índice y use el operador de asignación para establecer un nuevo valor.

int[] nums = {1, 2, 0, 4};  
// Change value at index 2  
nums[2] = 3;

### Index

An index refers to an element’s position within an array.

The index of an array starts from 0 and goes up to one less than the total length of the array.

int[] marks = {50, 55, 60, 70, 80};  
   
System.out.println(marks[0]);  
// Output: 50  
   
System.out.println(marks[4]);  
// Output: 80

### Arrays

In Java, an array is used to store a list of elements of the same datatype.

Arrays are fixed in size and their elements are ordered.

// Create an array of 5 int elements  
int[] marks = {10, 20, 30, 40, 50};

### Array creation in Java

In Java, an array can be created in the following ways:

* Using the {} notation, by adding each element all at once.
* Using the new keyword, and assigning each position of the array individually.

int[] age = {20, 21, 30};  
   
int[] marks = new int[3];  
marks[0] = 50;   
marks[1] = 70;  
marks[2] = 93;

### Changing an Element Value

### Para cambiar el valor de un elemento, seleccione el elemento a través de su índice y use el operador de asignación para establecer un nuevo valor.

int[] nums = {1, 2, 0, 4};  
// Change value at index 2  
nums[2] = 3;

### Java ArrayList

### En Java, se utiliza una ArrayList para representar una lista dinámica.

### Si bien las matrices de Java tienen un tamaño fijo (el tamaño no se puede modificar), una ArrayList permite flexibilidad al poder agregar y eliminar elementos.

// import the ArrayList package  
import java.util.ArrayList;  
  
// create an ArrayList called students  
ArrayList<String> students = new ArrayList<String>();

### Modificación de ArrayLists en Java

### Una ArrayList se puede modificar fácilmente utilizando métodos integrados.

### Para agregar elementos a una ArrayList, usa el método add (). El elemento que desea agregar va dentro de ().

### Para eliminar elementos de una ArrayList, usa el método remove (). Dentro de () puede especificar el índice del elemento que desea eliminar. Alternativamente, puede especificar directamente el elemento que desea eliminar.

import java.util.ArrayList;  
  
public class Students {  
  public static void main(String[] args) {  
      
     // create an ArrayList called studentList, which initially holds []  
        ArrayList<String> studentList = new ArrayList<String>();  
      
    // add students to the ArrayList  
    studentList.add("John");  
    studentList.add("Lily");  
    studentList.add("Samantha");  
    studentList.add("Tony");  
      
    // remove John from the ArrayList, then Lily  
    studentList.remove(0);  
    studentList.remove("Lily");  
      
    // studentList now holds [Samantha, Tony]  
      
    System.out.println(studentList);  
  }  
}

### Conceptos básicos de ArrayLists, que incluyen:

### Creando una ArrayList.

### Agregar un nuevo elemento ArrayList usando add ().

### Accediendo al tamaño de una ArrayList usando size ().

### Encontrar un elemento por índice usando get ().

### Cambiar el valor de un elemento ArrayList usando set ().

### Eliminar un elemento con un valor específico usando remove ().

### Recuperando el índice de un elemento con un valor específico usando indexOf ().

### Loops (Bucle)

### For-each declaración en Java

### En Java, la instrucción for-each le permite recorrer directamente cada elemento de una matriz o ArrayList y realizar alguna acción con cada elemento.

### Al crear una declaración for-each, debe incluir la palabra clave for y dos expresiones entre paréntesis, separadas por dos puntos. Éstas incluyen:

### El identificador de un elemento sobre el que estamos iterando actualmente.

### La matriz de origen o ArrayList sobre la que estamos iterando.

// array of numbers  
int[] numbers = {1, 2, 3, 4, 5};  
  
// for-each loop that prints each number in numbers  
// int num is the handle while numbers is the source array  
for (int num : numbers) {    
    System.out.println(num);  
}

**---- ---**

Un bucle **while** se parece un poco a una declaración **if**:

while (silliness > 10) {  
  // code to run  
}

Como una declaración **if**, el código dentro de un ciclo (loop) **while** solo se ejecutará si la condición es verdadera. Sin embargo, un ciclo **while** continuará ejecutando el código una y otra vez hasta que la condición se evalúe como falsa. Entonces, el bloque de código se repetirá hasta que la tontería sea menor o igual a 10.

// set attempts to 0  
int attempts = 0;  
// enter loop if condition is true  
while (passcode != 0524 && attempts < 4) {  
  System.out.println("Try again.");  
  passcode = getNewPasscode();  
  attempts += 1;  
  // is condition still true?  
  // if so, repeat code block  
}  
// exit when condition is not true

### Los bucles while son extremadamente útiles cuando desea ejecutar algún código hasta que ocurra un cambio específico. Sin embargo, si no está seguro de que se producirá un cambio, ¡tenga cuidado con el ciclo infinito!

### Se producen bucles infinitos cuando la condición nunca se evaluará como falsa. Esto puede hacer que todo el programa se bloquee.

### Para bucles

Incrementar con bucles es tan común en la programación que Java (como muchos otros lenguajes de programación) incluye una sintaxis específicamente para abordar este patrón: for loops.

Un encabezado de for *loop* se compone de las siguientes tres partes, cada una separada por un punto y coma:

### La inicialización de la variable de control de bucle.

### Una expresión booleana.

### Una declaración de incremento o decremento.

### La línea de apertura podría verse así:

for (int i = 0; i < 5; i++) {  
  
  // code that will run  
  
}

### En un for loop, se ejecuta una instrucción de inicialización una vez para inicializar la variable de control de bucle. Esta variable se modifica en cada iteración, se puede hacer referencia a ella en el cuerpo del bucle y se utiliza para probar la condición booleana. En el ejemplo anterior, i es la variable de control de bucle.

### Analicemos el ejemplo anterior:

### i = 0: i se inicializa a 0

### i <5: el bucle recibe una condición booleana que se basa en el valor de i. El ciclo continuará ejecutándose hasta que i <5 sea falso.

### i ++: incrementaré al final de cada ciclo y antes de que se reevalúe la condición.

Entonces, el código se ejecutará a través del ciclo un total de cinco veces.

También escucharemos el término "iteración" en referencia a los bucles. Cuando iteramos, solo significa que estamos repitiendo el mismo bloque de código.

Resumen

**while loops** (**bucles while):** son útiles para repetir un bloque de código un número desconocido de veces hasta que se cumpla alguna condición. Por ejemplo:

int wishes = 0;  
  
while (wishes < 3) {  
  
  // code that will run  
  wishes++;  
  
}

**for loops** (**bucles for)**: estos son ideales para cuando aumenta o disminuye con una variable de contador. Por ejemplo:

for (int i = 0; i < 5; i++) {  
  
  // code that will run  
  
}

**For-each loops** (Bucles para cada uno): facilitan hacer algo con cada elemento de una lista. Por ejemplo:

for (String inventoryItem : inventoryItems) {  
  
  // do something with each inventoryItem  
  
}

**STRING METHODS**

### length() String Method in Java length () Método de cadena en Java En Java, el método de cadena length () devuelve el número total de caracteres, la longitud, de una cadena.

String str = "Codecademy";    
  
System.out.println(str.length());  
// prints 10

### concat() String Method in Java

### En Java, el método de cadena concat () se usa para agregar una cadena al final de otra cadena. Este método devuelve una cadena que representa el texto de las cadenas combinadas.

String s1 = "Hello";  
String s2 = " World!";  
  
String s3 = s1.concat(s2);  
// concatenates strings s1 and s2  
  
System.out.println(s3);  
// prints "Hello World!"

### String Method equals() in Java

### En Java, el método de cadena equals () prueba la igualdad entre dos cadenas.

### equals () compara el contenido de cada String. Si todos los caracteres entre los dos coinciden, el método devuelve verdadero. Si alguno de los caracteres no coincide, devuelve falso.

### Además, si desea comparar dos cadenas sin considerar mayúsculas / minúsculas, puede usar .equalsIgnoreCase ().

String s1 = "Hello";  
String s2 = "World";  
  
System.out.println(s1.equals("Hello"));  
// prints true  
  
System.out.println(s2.equals("Hello"));  
// prints false   
  
System.out.println(s2.equalsIgnoreCase("world"));  
// prints true

### indexOf() String Method in Java

### En Java, el método de cadena indexOf () devuelve la primera aparición de un carácter o una subcadena en una cadena. El carácter / subcadena cuyo índice desea encontrar va dentro de ().

### Si indexOf () no puede encontrar el carácter o la subcadena, devolverá -1.

String str = "Hello World!";  
  
System.out.println(str.indexOf("l"));  
// prints 2  
  
System.out.println(str.indexOf("Wor"));  
// prints 6  
  
System.out.println(str.indexOf("z"));  
// prints -1

### charAt() String Method in Java

### En Java, el método de cadena charAt () devuelve el carácter de una cadena en un índice especificado. El valor del índice se pasa dentro de () y debe estar entre 0 y length () - 1.

String str = "This is a string";  
  
System.out.println(str.charAt(0));  
// prints 'T'  
  
System.out.println(str.charAt(15));  
// prints 'g'

### toUpperCase() and toLowerCase() String Methods

### En Java, podemos convertir fácilmente una cadena a mayúsculas y minúsculas con la ayuda de algunos métodos de cadena:

### toUpperCase () devuelve el valor de cadena convertido a mayúsculas.

### toLowerCase () devuelve el valor de cadena convertido a minúsculas.

String str = "Hello World!";  
  
String uppercase = str.toUpperCase();  
// uppercase = "HELLO WORLD!"  
  
String lowercase = str.toLowerCase();  
// lowercase = "hello world!"

substring()

Puede haber ocasiones en las que solo queramos una parte de una cadena. En tales casos, es posible que deseemos extraer una subcadena de una cadena.

El método substring () hace exactamente eso. Por ejemplo:

String line = "The Heav'ns and all the Constellations rung";  
  
System.out.println(line.substring(24));

### Produciría el renglón de Constelaciones porque eso es lo que comienza en el índice 24 y termina al final de la línea. La subcadena comienza con el carácter en el índice especificado y se extiende hasta el final de la cadena.

### Pero supongamos que queremos una subcadena del medio de la cadena. Podemos incluir dos argumentos con este método de cadena. Por ejemplo:

String line = "The Heav'ns and all the Constellations rung";  
  
System.out.println(line.substring(24, 38));

### Produciría Constelaciones porque esa es la subcadena que comienza en el índice 24 y termina en el índice 38.

### Podemos usar este método para devolver una subcadena de un solo elemento en un índice específico llamando a substring () con el valor del índice deseado como primer argumento y luego el valor del índice más uno como segundo argumento.

### Por ejemplo, podemos usar este método para generar solo C:

String line = "The Heav'ns and all the Constellations rung";  
  
System.out.println(line.substring(24, 25));  
// Prints: C

### Access, Encapsulation, and Static Methods

### The public and private keywords

### En Java, las palabras clave public y private definen el acceso de clases, variables de instancia, constructores y métodos.

### private restringe el acceso solo a la clase que declaró la estructura, mientras que public permite el acceso desde cualquier clase.

### Encapsulation La encapsulación es una técnica utilizada para mantener los detalles de implementación ocultos a otras clases. Su objetivo es crear pequeños paquetes de lógica.

### The private Keyword

### En Java, las variables de instancia se encapsulan mediante la palabra clave private. Esto evita que otras clases accedan directamente a estas variables.

public class CheckingAccount{  
  // Three private instance variables  
  private String name;  
  private int balance;  
  private String id;  
}

### Accessor Methods

### En Java, los métodos de acceso devuelven el valor de una variable privada. Esto le da a otras clases acceso a ese valor almacenado en esa variable. sin tener acceso directo a la propia variable.

### Los métodos de acceso no toman parámetros y tienen un tipo de retorno que coincide con el tipo de variable a la que están accediendo.

public class CheckingAccount{  
  private int balance;  
    
  //An accessor method  
  public int getBalance(){  
    return this.balance;  
  }  
}

### Mutator Methods

### En Java, los métodos mutadores restablecen el valor de una variable privada. Esto le da a otras clases la capacidad de modificar el valor almacenado en esa variable sin tener acceso directo a la variable en sí.

### Los métodos mutadores toman un parámetro cuyo tipo coincide con el tipo de variable que está modificando. Los métodos mutantes generalmente no devuelven nada.

public class CheckingAccount{  
  private int balance;  
    
  //A mutator method  
  public void setBalance(int newBalance){  
    this.balance = newBalance;  
  }  
}

### Local Variables En Java, las variables locales solo se pueden usar dentro del alcance en el que se definieron. Este alcance a menudo se define mediante un conjunto de llaves. Las variables no se pueden usar fuera de esos corchetes.

public void exampleMethod(int exampleVariable){  
  // exampleVariable can only be used inside these curly brackets.  
}

### The this Keyword with Variables

### En Java, la palabra clave this se puede utilizar para designar la diferencia entre las variables de instancia y las variables locales. Variables con this. hacer referencia a una variable de instancia.

public class Dog{  
  public String name;  
  
  public void speak(String name){  
    // Prints the instance variable named name  
    System.out.println(this.name);  
  
    // Prints the local variable named name  
    System.out.println(name);  
  }  
}

### The this Keyword with Methods

### En Java, la palabra clave this se puede utilizar para llamar a métodos al escribir clases.

public class ExampleClass{  
  public void exampleMethodOne(){  
    System.out.println("Hello");  
  }  
  
  public void exampleMethodTwo(){  
    //Calling a method using this.  
    this.exampleMethodOne();  
    System.out.println("There");  
  }  
}

### Static Methods

### Los métodos estáticos son métodos que se pueden llamar dentro de un programa sin crear un objeto de la clase.

// static method  
public static int getTotal(int a, int b) {  
  return a + b;  
}  
  
public static void main(String[] args) {  
  int x = 3;  
  int y = 2;  
  System.out.println(getTotal(x,y)); // Prints: 5  
}

### Calling a Static Method

### Los métodos estáticos se pueden llamar agregando el operador de punto a un nombre de clase seguido del nombre del método.

int largerNumber = Math.max(3, 10); // Call static method  
System.out.println(largerNumber); // Prints: 10

### The Math Class La clase Math (que es parte del paquete java.lang) contiene una variedad de métodos estáticos que se pueden usar para realizar cálculos numéricos.

System.out.println(Math.abs(-7.0)); // Prints: 7  
  
System.out.println(Math.pow(5, 3)); // Prints: 125.0  
  
System.out.println(Math.sqrt(52)); // Prints: 7.211102550927978

### The public and private keywords

In Java, the keywords public and private define the access of classes, instance variables, constructors, and methods.

private restricts access to only the class that declared the structure, while public allows for access from any class.

### Encapsulation

Encapsulation is a technique used to keep implementation details hidden from other classes. Its aim is to create small bundles of logic.

### The private Keyword

In Java, instance variables are encapsulated by using the private keyword. This prevents other classes from directly accessing these variables.

public class CheckingAccount{  
  // Three private instance variables  
  private String name;  
  private int balance;  
  private String id;  
}

### Accessor Methods

In Java, accessor methods return the value of a private variable. This gives other classes access to that value stored in that variable. without having direct access to the variable itself.

Accessor methods take no parameters and have a return type that matches the type of the variable they are accessing.

public class CheckingAccount{  
  private int balance;  
    
  //An accessor method  
  public int getBalance(){  
    return this.balance;  
  }  
}

### Mutator Methods

In Java, mutator methods reset the value of a private variable. This gives other classes the ability to modify the value stored in that variable without having direct access to the variable itself.

Mutator methods take one parameter whose type matches the type of the variable it is modifying. Mutator methods usually don’t return anything.

public class CheckingAccount{  
  private int balance;  
    
  //A mutator method  
  public void setBalance(int newBalance){  
    this.balance = newBalance;  
  }  
}

### Local Variables

In Java, local variables can only be used within the scope that they were defined in. This scope is often defined by a set of curly brackets. Variables can’t be used outside of those brackets.

public void exampleMethod(int exampleVariable){  
  // exampleVariable can only be used inside these curly brackets.  
}

### The this Keyword with Variables

In Java, the this keyword can be used to designate the difference between instance variables and local variables. Variables with this. reference an instance variable.

public class Dog{  
  public String name;  
   
  public void speak(String name){  
    // Prints the instance variable named name  
    System.out.println(this.name);  
   
    // Prints the local variable named name  
    System.out.println(name);  
  }  
}

### The this Keyword with Methods

In Java, the this keyword can be used to call methods when writing classes.

public class ExampleClass{  
  public void exampleMethodOne(){  
    System.out.println("Hello");  
  }  
   
  public void exampleMethodTwo(){  
    //Calling a method using this.  
    this.exampleMethodOne();  
    System.out.println("There");  
  }  
}

### Static Methods

Static methods are methods that can be called within a program without creating an object of the class.

// static method  
public static int getTotal(int a, int b) {  
  return a + b;  
}  
   
public static void main(String[] args) {  
  int x = 3;  
  int y = 2;  
  System.out.println(getTotal(x,y)); // Prints: 5  
}

### Calling a Static Method

Static methods can be called by appending the dot operator to a class name followed by the name of the method.

int largerNumber = Math.max(3, 10); // Call static method  
System.out.println(largerNumber); // Prints: 10

### The Math Class

The Math class (which is part of the java.lang package) contains a variety of static methods that can be used to perform numerical calculations.

System.out.println(Math.abs(-7.0)); // Prints: 7  
   
System.out.println(Math.pow(5, 3)); // Prints: 125.0  
   
System.out.println(Math.sqrt(52)); // Prints: 7.211102550927978

### The static Keyword Los métodos y variables estáticos se declaran como estáticos mediante el uso de la palabra clave estática en la declaración.

public class ATM{  
  // Static variables  
  public static int totalMoney = 0;  
  public static int numATMs = 0;  
  
  // A static method  
  public static void averageMoney(){  
    System.out.println(totalMoney / numATMs);  
  }

### Static Methods and Variables

### Los métodos y variables estáticos se asocian con la clase como un todo, no con los objetos de la clase. Ambos se utilizan mediante el nombre de la clase seguido del. operador.

public class ATM{  
  // Static variables  
  public static int totalMoney = 0;  
  public static int numATMs = 0;  
  
  // A static method  
  public static void averageMoney(){  
    System.out.println(totalMoney / numATMs);  
  }  
  
  public static void main(String[] args){  
  
    //Accessing a static variable  
    System.out.println("Total number of ATMs: " + ATM.numATMs);   
  
    // Calling a static method  
    ATM.averageMoney();  
  }  
  
}

### The public and private keywords

In Java, the keywords public and private define the access of classes, instance variables, constructors, and methods.

private restricts access to only the class that declared the structure, while public allows for access from any class.

### Encapsulation

Encapsulation is a technique used to keep implementation details hidden from other classes. Its aim is to create small bundles of logic.

### The private Keyword

In Java, instance variables are encapsulated by using the private keyword. This prevents other classes from directly accessing these variables.

public class CheckingAccount{  
  // Three private instance variables  
  private String name;  
  private int balance;  
  private String id;  
}

### Accessor Methods

In Java, accessor methods return the value of a private variable. This gives other classes access to that value stored in that variable. without having direct access to the variable itself.

Accessor methods take no parameters and have a return type that matches the type of the variable they are accessing.

public class CheckingAccount{  
  private int balance;  
    
  //An accessor method  
  public int getBalance(){  
    return this.balance;  
  }  
}

### Mutator Methods

In Java, mutator methods reset the value of a private variable. This gives other classes the ability to modify the value stored in that variable without having direct access to the variable itself.

Mutator methods take one parameter whose type matches the type of the variable it is modifying. Mutator methods usually don’t return anything.

public class CheckingAccount{  
  private int balance;  
    
  //A mutator method  
  public void setBalance(int newBalance){  
    this.balance = newBalance;  
  }  
}

### Local Variables

In Java, local variables can only be used within the scope that they were defined in. This scope is often defined by a set of curly brackets. Variables can’t be used outside of those brackets.

public void exampleMethod(int exampleVariable){  
  // exampleVariable can only be used inside these curly brackets.  
}

### The this Keyword with Variables

In Java, the this keyword can be used to designate the difference between instance variables and local variables. Variables with this. reference an instance variable.

public class Dog{  
  public String name;  
   
  public void speak(String name){  
    // Prints the instance variable named name  
    System.out.println(this.name);  
   
    // Prints the local variable named name  
    System.out.println(name);  
  }  
}

### The this Keyword with Methods

In Java, the this keyword can be used to call methods when writing classes.

public class ExampleClass{  
  public void exampleMethodOne(){  
    System.out.println("Hello");  
  }  
   
  public void exampleMethodTwo(){  
    //Calling a method using this.  
    this.exampleMethodOne();  
    System.out.println("There");  
  }  
}

### Static Methods

Static methods are methods that can be called within a program without creating an object of the class.

// static method  
public static int getTotal(int a, int b) {  
  return a + b;  
}  
   
public static void main(String[] args) {  
  int x = 3;  
  int y = 2;  
  System.out.println(getTotal(x,y)); // Prints: 5  
}

### Calling a Static Method

Static methods can be called by appending the dot operator to a class name followed by the name of the method.

int largerNumber = Math.max(3, 10); // Call static method  
System.out.println(largerNumber); // Prints: 10

### The Math Class

The Math class (which is part of the java.lang package) contains a variety of static methods that can be used to perform numerical calculations.

System.out.println(Math.abs(-7.0)); // Prints: 7  
   
System.out.println(Math.pow(5, 3)); // Prints: 125.0  
   
System.out.println(Math.sqrt(52)); // Prints: 7.211102550927978

### The static Keyword

Static methods and variables are declared as static by using the statickeyword upon declaration.

public class ATM{  
  // Static variables  
  public static int totalMoney = 0;  
  public static int numATMs = 0;  
   
  // A static method  
  public static void averageMoney(){  
    System.out.println(totalMoney / numATMs);  
  }

### Static Methods and Variables

Static methods and variables are associated with the class as a whole, not objects of the class. Both are used by using the name of the class followed by the . operator.

public class ATM{  
  // Static variables  
  public static int totalMoney = 0;  
  public static int numATMs = 0;  
   
  // A static method  
  public static void averageMoney(){  
    System.out.println(totalMoney / numATMs);  
  }  
   
  public static void main(String[] args){  
   
    //Accessing a static variable  
    System.out.println("Total number of ATMs: " + ATM.numATMs);   
   
    // Calling a static method  
    ATM.averageMoney();  
  }  
   
}

### Static Methods with Instance Variables

### Los métodos estáticos no pueden acceder ni cambiar los valores de las variables de instancia.

class ATM{  
// Static variables  
  public static int totalMoney = 0;  
  public static int numATMs = 0;   
  
  public int money = 1;  
  
  // A static method  
  public static void averageMoney(){  
    // Can not use this.money here because a static method can't access instance variables  
  }  
  
}

### Methods with Static Variables Tanto los métodos estáticos como los no estáticos pueden acceder o cambiar los valores de las variables estáticas.

class ATM{  
// Static variables   
  public static int totalMoney = 0;   
  public static int numATMs = 0;   
  public int money = 1;  
  
// A static method interacting with a static variable   
  public static void staticMethod(){   
    totalMoney += 1;  
   }   
  
  // A non-static method interactingwith a static variable   
  public void nonStaticMethod(){  
    totalMoney += 1;   
  }   
}

### Static Methods and the this Keyword

### Los métodos estáticos no tienen esta referencia y, por lo tanto, no pueden usar las variables de instancia de la clase o llamar a métodos no estáticos.

public class DemoClass{  
  
  public int demoVariable = 5;  
  
  public void demoNonStaticMethod(){  
      
  }  
  public static void demoStaticMethod(){  
    // Can't use "this.demoVariable" or "this.demoNonStaticMethod()"  
  }  
}

**Inheritance and Polymorphism** (Herencia y polimorfismo)

Herencia en Java

La herencia es una característica importante de la programación orientada a objetos en Java. Permite que una clase (clase secundaria) herede los campos y métodos de otra clase (clase principal). Por ejemplo, podríamos querer un perro de clase secundaria para los rasgos inherentes de un animal de clase padre más general.

Al definir una clase secundaria en Java, usamos la palabra clave extiende para heredar de una clase principal.

// Parent Class  
class Animal {  
  // Animal class members  
}  
  
// Child Class  
class Dog extends Animal {  
  // Dog inherits traits from Animal   
    
  // additional Dog class members  
}

**Método Main () en Java**

En programas simples de Java, puede trabajar con una sola clase y un archivo. Sin embargo, a medida que sus programas se vuelven más complejos, trabajará con varias clases, cada una de las cuales requiere su propio archivo. Solo uno de estos archivos en el paquete Java requiere un método main (), y este es el archivo que se ejecutará en el paquete.

Por ejemplo, digamos que tenemos dos archivos en nuestro paquete Java para dos clases diferentes:

Shape, la clase parent

Square, la clase child

Si el archivo Java que contiene nuestra clase Shape es el único con un método main (), este es el archivo que se ejecutará para nuestro paquete Java.

// Shape.java file   
class Shape {  
  public static void main(String[] args) {  
    Square sq = new Square();  
  }  
}  
  
// Square.java file   
class Square extends Shape {  
    
}

### super() in Java

En Java, una clase secundaria hereda los campos y métodos de su padre, lo que significa que también hereda el constructor del padre. A veces, es posible que deseemos modificar el constructor, en cuyo caso podemos usar el método super (), que actúa como el constructor padre dentro del constructor de la clase secundaria.

Alternativamente, también podemos anular completamente un constructor de clase parent escribiendo un nuevo constructor para la clase child.

// Parent class  
class Animal {  
  String sound;  
  Animal(String snd) {  
    this.sound = snd;  
  }  
}  
  
// Child class  
class Dog extends Animal {   
  // super() method can act like the parent constructor inside the child class constructor.  
  Dog() {  
    super("woof");  
  }   
  // alternatively, we can override the constructor completely by defining a new constructor.  
  Dog() {  
    this.sound = "woof";  
  }  
}

Palabras clave protegidas y finales en Java

Al crear clases en Java, a veces es posible que deseemos controlar el acceso de la clase secundaria a los miembros de la clase principal. Podemos usar las palabras clave protected and final para hacer precisamente eso.

protected mantiene un miembro de la clase padre accesible a sus clases child, a archivos dentro de su propio paquete y por subclases de esta clase en otro paquete.

Agregar final antes del modificador de acceso del método de la clase principal hace que las clases secundarias no puedan modificar ese método; es inmutable.

class Student {  
  protected double gpa;  
  // any child class of Student can access gpa   
    
  final protected boolean isStudent() {  
    return true;  
  }  
  // any child class of Student cannot modify isStudent()  
}

### Polymorphism in Java

Java incorpora el principio de polimorfismo de programación orientada a objetos.

El polimorfismo permite que una clase secundaria comparta la información y el comportamiento de su clase principal al tiempo que incorpora su propia funcionalidad. Esto permite los beneficios de una sintaxis simplificada y una sobrecarga cognitiva reducida para los desarrolladores.

// Parent class  
class Animal {  
  public void greeting() {  
    System.out.println("The animal greets you.");  
  }  
}  
  
// Child class  
class Cat extends Animal {  
  public void greeting() {  
    System.out.println("The cat meows.");  
  }  
}  
  
class MainClass {  
  public static void main(String[] args) {  
    Animal animal1 = new Animal();  // Animal object  
    Animal cat1 = new Cat();  // Cat object  
    animal1.greeting(); // prints "The animal greets you."  
    cat1.greeting(); // prints "The cat meows."  
  }  
}

### Method Overriding in Java

### En Java, podemos anular fácilmente los métodos de la clase principal en una clase secundaria. Anular un método es útil cuando queremos que nuestro método de clase secundaria tenga el mismo nombre que un método de clase padre pero se comporte un poco diferente.

### Para anular un método de clase principal en una clase secundaria, debemos asegurarnos de que el método de clase secundaria tenga lo siguiente en común con su método de clase principal:

### Nombre del método

### Tipo de retorno

### Número y tipo de parámetros

### Además, deberíamos incluir la palabra clave @Override encima de nuestro método de clase secundaria para indicar al compilador que queremos anular un método en la clase principal.

// Parent class   
class Animal {  
  public void eating() {  
    System.out.println("The animal is eating.");  
  }  
}  
  
// Child class   
class Dog extends Animal {  
  // Dog's eating method overrides Animal's eating method  
    @Override  
  public void eating() {  
    System.out.println("The dog is eating.");  
  }  
}

### Child Classes in Arrays and ArrayLists

### En Java, el polimorfismo nos permite poner instancias de diferentes clases que comparten una clase padre juntas en una matriz o ArrayList.

### Por ejemplo, si tenemos una clase padre Animal con clases secundarias Gato, Perro y Cerdo, podemos configurar una matriz con instancias de cada animal y luego recorrer la lista de animales para realizar la misma acción en cada uno.

// Animal parent class with child classes Cat, Dog, and Pig.   
Animal cat1, dog1, pig1;  
  
cat1 = new Cat();  
dog1 = new Dog();  
pig1 = new Pig();  
  
// Set up an array with instances of each animal  
Animal[] animals = {cat1, dog1, pig1};  
  
// Iterate through the list of animals and perform the same action with each  
for (Animal animal : animals) {  
    
  animal.sound();  
    
}

### Sololearn.com

**Conditional Statements**

# Decision Making

**Conditional statements** are used to perform different actions based on different conditions.   
The**if statement** is one of the most frequently used conditional statements.   
If the **if**statement's condition expression evaluates to true, the block of code inside the **if**statement is executed. If the expression is found to be false, the first set of code after the end of the **if**statement (after the closing curly brace) is executed.   
**Syntax:**

**if** (condition) {   
//Executes when the condition is true   
}

**JAVA**

Any of the following comparison operators may be used to form the condition:   
**<**less than   
**>**greater than   
**!=** not equal to   
**==**equal to   
**<=**less than or equal to   
**>=**greater than or equal to   
  
**For example:**

int x = 7;   
**if**(x < 42) {   
System.out.println("Hi");   
}

**JAVA**

Try it Yourself

Remember that you need to use two equal signs (**==**) to test for equality, since a single equal sign is the assignment operator.

Nested if Statements (Declaraciones if anidadas)  
You can use one **if-else** statement inside another**if** or **else**statement.   
**For example:**

int age = 25;   
if(age > 0) {   
if(age > 16) {   
System.out.println("Welcome!");   
} else {   
System.out.println("Too Young");   
}   
} else {   
System.out.println("Error");   
}

**JAVA**

Try it Yourself

You can nest as many **if-else** statements as you want.

# else if Statements

Instead of using nested **if-else** statements, you can use the **else if** statement to check multiple conditions.

**For example:**

int age = 25;   
  
**if**(age <= 0) {   
System.out.println("Error");   
} **else if**(age <= 16) {   
System.out.println("Too Young");   
} **else if**(age < 100) {   
System.out.println("Welcome!");   
} **else** {   
System.out.println("Really?");   
}

**JAVA**

Try it Yourself

The code will check the condition to evaluate to true and execute the statements inside that block.

You can include as many **else if** statements as you need.

# Logical Operators Logical operators are used to combine multiple conditions. Let's say you wanted your program to output "Welcome!" only when the variable **age**is greater than 18 and the variable **money**is greater than 500. One way to accomplish this is to use nested **if**statements:

if (age > 18) {   
if (money > 500) {   
System.out.println("Welcome!");   
}   
}

**JAVA**

Try it Yourself

However, using the **AND**logical operator (**&&**) is a better way:

if (age > 18 **&&** money > 500) {   
System.out.println("Welcome!");   
}

**JAVA**

Try it Yourself

If both operands of the AND operator are true, then the condition becomes true.

# The switch Statement

A **switch**statement tests a variable for equality against a list of values. Each value is called a **case**, and the variable being switched on is checked for each case.   
**Syntax:**

**switch** (expression) {   
**case** value1 :   
//Statements   
**break**; //optional   
**case** value2 :   
//Statements   
**break**; //optional   
//You can have any number of case statements.   
**default** : //Optional   
//Statements   
}

**JAVA**

- When the variable being switched on is equal to a **case**, the statements following that **case**will execute until a **break**statement is reached.   
- When a **break**statement is reached, the **switch**terminates, and the flow of control jumps to the next line after the **switch**statement.   
- Not every **case**needs to contain a **break**. If no **break**appears, the flow of control will fall through to subsequent cases until a **break**is reached.   
  
The example below tests **day**against a set of values and prints a corresponding message.

int day = 3;   
  
**switch**(day) {   
**case** 1:   
System.out.println("Monday");   
break;   
**case** 2:   
System.out.println("Tuesday");   
break;   
**case** 3:   
System.out.println("Wednesday");   
break;   
}

**JAVA**

Try it Yourself

You can have any number of **case**statements within a **switch**. Each **case**is followed by the comparison value and a colon.

# while Loops A **loop**statement allows to repeatedly execute a statement or group of statements. A **while**loop statement repeatedly executes a target statement as long as a given condition is true. **Example:**

int x = 3;   
  
**while**(x > 0) {   
System.out.println(x);   
x--;   
}

**JAVA**

Try it Yourself

The **while**loops check for the condition x > 0. If it evaluates to true, it executes the statements within its body. Then it checks for the statement again and repeats.

Notice the statement x--. This decrements x each time the loop runs, and makes the loop stop when x reaches 0.   
Without the statement, the loop would run forever.

# for Loops

Another loop structure is the **for**loop. A for loop allows you to efficiently write a loop that needs to execute a specific number of times.   
**Syntax:**

**for** (initialization; condition; increment/decrement) {   
statement(s)   
}

**JAVA**

**Initialization**: Expression executes only once during the beginning of loop   
**Condition**: Is evaluated each time the loop iterates. The loop executes the statement repeatedly, until this condition returns false.   
**Increment/Decrement**: Executes after each iteration of the loop.   
  
The following example prints the numbers 1 through 5.

for(int x = 1; x <=5; x++) {   
System.out.println(x);   
}

**JAVA**

Try it Yourself

This initializes x to the value 1, and repeatedly prints the value of x, until the condition x<=5 becomes false. On each iteration, the statement x++ is executed, incrementing x by one.

Notice the semicolon (;) after initialization and condition in the syntax.

# do...while Loops

A **do...while** loop is similar to a **while**loop, except that a **do...while** loop is guaranteed to execute at least one time.   
**Example:**

int x = 1;   
**do** {   
System.out.println(x);   
x++;   
} **while**(x < 5);   
  
/\*   
1   
2   
3   
4   
\*/

**JAVA**

Try it Yourself

Notice that the condition appears at the end of the loop, so the statements in the loop execute once before it is tested.   
Even with a false condition, the code will run once.   
**Example:**

int x = 1;   
**do** {   
System.out.println(x);   
x++;   
} **while**(x < 0);

**JAVA**

Try it Yourself

Notice that in do…while loops, the while is just the condition and doesn't have a body itself.

# Arrays

An **array**is a collection of variables of the same type.   
When you need to store a list of values, such as numbers, you can store them in an array, instead of declaring separate variables for each number.   
  
To declare an array, you need to define the type of the elements with **square brackets**.   
**For example, to declare an array of integers:**

**int[ ]** arr;

**JAVA**

The name of the array is **arr**. The type of elements it will hold is **int**.   
  
Now, you need to define the array's capacity, or the number of elements it will hold. To accomplish this, use the keyword **new**

int[ ] arr = **new** int[5];

**JAVA**

The code above declares an array of 5 integers.   
In an array, the elements are ordered and each has a specific and constant position, which is called an **index**.   
  
To reference elements in an array, type the name of the array followed by the index position within a pair of square brackets.   
**Example:**

**arr[2]** = 42;

**JAVA**

This assigns a value of 42 to the element with 2 as its index.

Note that elements in the array are identified with **zero-based** index numbers, meaning that the first element's index is 0 rather than one. So, the maximum index of the array int[5] is 4.

# Array Length

You can access the length of an array (the number of elements it stores) via its **length** property.   
**Example:**

int[ ] intArr = new int[5];   
System.out.println(**intArr.length**);

**JAVA**

Try it Yourself

Don't forget that in arrays, indexes start from 0. So, in the example above, the last index is 4.

# Enhanced for Loop

The**enhanced for loop** (sometimes called a "for each" loop) is used to traverse elements in arrays.   
The advantages are that it eliminates the possibility of bugs and makes the code easier to read.   
**Example:**

int[ ] primes = {2, 3, 5, 7};   
  
**for (int t: primes)** {   
System.out.println(t);   
}   
  
/\*   
2   
3   
5   
7   
\*/

**JAVA**

Try it Yourself

The**enhanced for loop** declares a variable of a type compatible with the elements of the array being accessed. The variable will be available within the **for**block, and its value will be the same as the current array element.   
So, on each iteration of the loop, the variable **t**will be equal to the corresponding element in the array.

Notice the **colon**after the variable in the syntax.

# Multidimensional Arrays

**Multidimensional**arrays are array that contain other arrays. The two-dimensional array is the most basic multidimensional array.   
To create multidimensional arrays, place each array within its own set of square brackets.   
**Example of a two-dimensional array:**

**int[ ][ ]** sample = { {1, 2, 3}, {4, 5, 6} };

**JAVA**

This declares an array with two arrays as its elements.   
To access an element in the two-dimensional array, provide two indexes, one for the array, and another for the element inside that array.   
The following example accesses the first element in the second array of sample.

int x = sample**[1][0]**;   
System.out.println(x);

**JAVA**

Try it Yourself

The array's two indexes are called **row index** and **column index**.